Learning to Score Final Positions in the Game of Go

Erik C.D. van der Werf [*], H. Jaap van den Herik,
Jos W.H.M. Uiterwijk

Department of Computer Science, Institute for Knowledge and Agent Technology, Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands

Abstract

This article investigates the application of machine-learning techniques for the task of scoring final positions in the game of Go. Neural network classifiers are trained to classify life and death from labelled 9×9 game records. The performance is compared to standard classifiers from statistical pattern recognition. A recursive framework for classification is used to improve performance iteratively. Using a maximum of 4 iterations our Cascaded Scoring Architecture (CSA*) scores 98.9% of the positions correctly. Nearly all incorrectly scored positions are recognised (they can be corrected by a human operator). By providing reliable score information CSA* opens the large source of Go knowledge implicitly available in human game records for automatic extraction. It thus paves the way for a successful application of machine learning in Go.

Key words: Go, learning, neural net, scoring, game records, life and death

1 Introduction

Evaluating Go [1] positions is one of the hardest tasks in Artificial Intelligence (AI) [2,3]. In the last decades Go has received significant attention from AI research [4,5]. This was stimulated by Ing's million-dollar prize for the first computer program to defeat a professional Go player (it has expired unchallenged). Yet, despite all efforts, the best computer Go programs are still no match even for human amateurs of only

[*] Corresponding author Tel.: (+31) (0)43-38 83491; fax: (+31) (0)43-38 84897
Email addresses: `E.vanderWerf@cs.unimaas.nl` (Erik C.D. van der Werf), `herik@cs.unimaas.nl` (H. Jaap van den Herik), `uiterwijk@cs.unimaas.nl` (Jos W.H.M. Uiterwijk).

[1] For general information about the game including an introduction to the rules readers are advised to visit gobase.org [1].

moderate skill. Partially this is due to the complexity of Go, which makes brute-force search techniques infeasible on the 19×19 board. However, on the 9×9 board, which has a complexity between Chess and Othello [4], the *current* Go programs perform nearly as bad. The main reason lies in the lack of good positional evaluation functions. Many (if not all) of the current top programs rely on (huge) static knowledge bases derived from the programmers' Go skills and Go knowledge. As a consequence the top programs are extremely complex and difficult to improve. In principle a learning system should be able to overcome this problem.

In the past decade several researchers have used machine-learning techniques in Go. After Tesauro's [6] success story many researchers, including Dahl [7], Enzenberger [8] and Schraudolph et al. [9], have applied Temporal Difference (TD) learning for learning evaluation functions. Although TD-learning is a promising technique, which was underlined by NEUROGO's silver medal in 9×9 Go at the 8th Computer Olympiad in Graz [10], there has not been a major breakthrough, such as in Backgammon, and we believe that this will remain unlikely to happen in the near future as long as most learning is done from self-play or against weak opponents.

Over centuries humans have acquired extensive knowledge of Go. Since the knowledge is implicitly available in the games of human experts, it should be possible to apply machine-learning techniques to extract that knowledge from game records. So far, game records have only been used successfully for move prediction [7,11,12]. However, we are convinced that much more can be learned from these game records.

One of the best sources of game records on the Internet is the No Name Go Server game archive [13]. NNGS is a free on-line Go club where people from all over the world can meet and play Go. All games played on NNGS since 1995 are available on-line. Although NNGS game records contain a wealth of information, the automated extraction of knowledge from these games is a non-trivial task at least for the following three reasons.

Missing Information. Life-and-death status of blocks is not available. In scored games only a single numeric value representing the difference in points is available.

Unfinished Games. Not all games are scored. Human games often end by one side resigning or abandoning the game without finishing it, which often leaves the status of large parts of the board unclear. [2]

Bad Moves. During the game mistakes are made which are hard to detect. Since mistakes break the chain of optimal moves it can be misleading (and incorrect from a game-theoretical point of view) to relate positions before the mistake to the final outcome of the game.

[2] In professional games that are not played on-line similar problems can occur when the final reinforcing moves are omitted because they are considered obvious.

The first step towards making the knowledge in the game records accessible is to obtain reliable scores at the end of the game. Reliable scores require correct classifications of life and death. This article focuses on determining life and death for final positions. By focusing on final positions we avoid the problem of unfinished games and bad moves during the game, which will have to be dealt with later.

It has been pointed out by Müller [14] that *proving* the score of final positions is a hard task. For a set of typical human final positions, Müller showed that extending Benson's techniques for proving life and death [15] with a more sophisticated static analysis and search, still leaves around 75% of the board points unproven. His program EXPLORER classified most blocks correctly, but did so by means of a heuristic classification. Moreover, it still left some regions unsettled (they should be played out further). Although this may be appropriate for computer-computer games, it can be annoying in human-computer games, especially under the Japanese rules which penalise playing more stones than necessary.

Since *proving* the score of most final positions is not (yet) an option, we focus on learning a heuristic classification. In this article our main focus is on learning the heuristic classification by a multi-layer perceptron (MLP). Consequently, the heuristic rules that are learned for the classification will be implicitly contained in the numerical weights of the MLP, which are not easily understood by humans. However, it may be possible to use our framework to train alternative classifiers, such as a decision tree classifier, which could facilitate the extraction of heuristic rules that are more easily understood by humans. The next step is then to formulate the heuristics as lemmas, i.e., formulae which can be proven by using domain specific knowledge. As soon as a classification by lemmas is possible, scoring final positions is performed provably correct. This technique is adopted from the chess endgame two knights against pawn [16].

We believe that a learning algorithm for scoring final positions is important because: (1) it provides a more flexible framework than the traditional hand-coded static knowledge bases, and (2) it is a necessary first step towards learning to evaluate non-final positions. In general such an algorithm is good to have because: (1) large numbers of game records are hard to score manually, (2) publicly available programs still make too many mistakes scoring final positions, and (3) it can avoid unnecessarily long human-computer games.

The remainder of this article is organised as follows. Section 2 discusses the scoring method. Section 3 presents the learning task. Section 4 introduces the representation. Section 5 provides details about the dataset. Section 6 reports our experiments. Finally, section 7 presents our conclusion and on-going work.

2 The Scoring Method

The two main scoring methods in Go are territory scoring and area scoring. Territory scoring, used by the Japanese rules, counts the surrounded territory plus the number of captured opponent stones. Area scoring, used by the Chinese rules, counts the surrounded territory plus the alive stones on the board. The result of the two methods is usually the same up to one point. The result may differ since one player placed more stones than the other, for three possible reasons; (1) because Black made the first and the last move, (2) because one side passed more often during the game, and (3) because of handicap stones. Under Japanese rules the score may also differ because territory surrounded by alive stones in seki is not counted. In this article, area scoring is used since it is the simplest scoring method to implement for computers.

Area scoring works as follows. First, the life-and-death status of blocks of connected stones is determined. Second, dead stones are removed from the board. Third, each empty point is marked Black, White, or neutral. The non-empty points are already marked by their colour. The empty points can be marked by flood filling or by distance. Flood filling recursively marks empty points to their adjacent colour. In the case that a flood fill for Black overlaps with a flood fill for White the overlapping region becomes neutral. (As a consequence all non-neutral empty regions must be completely enclosed by one colour.) Scoring by distance marks each point based on the distance towards the nearest remaining black or white stone(s). If the point is closer to a black stone it is marked black, if the point is closer to a white stone it is marked white, otherwise (if the distance is equal) the point does not affect the score and is marked neutral. Finally, the difference between black and white points, together with a possible komi, determines the outcome of the game.

In final positions scoring by flood filling and scoring by distance should give the same result. If the result is not the same, there are large open regions with unsettled interior points, which usually means that some stones should have been removed or some points could still be gained by playing further. Comparing flood filling with scoring by distance is therefore a useful check to detect whether the game is finished and scored correctly.

3 The Learning Task

The task of learning to score comes down to learning to determine which blocks of connected stones are dead and should be removed from the board. This is learned from a set of labelled final positions, for which the labels contain the colour controlling each point. A straightforward implementation would be to learn classifying all blocks based on the labelled points. However, for some blocks this is not a good idea because their status can be irrelevant and forcing them to be classified just complicates the learning task.

3.1 Which Blocks to Classify?

For arriving at a correct score we require correct classifications for only two types of blocks. The first type is dead in the opponent's area. The second type is alive and at the border of friendly area. (Notice that, for training, the knowledge where the border is will be obtained from labelled game records.) The distinction between block types is illustrated in Figure 1. Here all marked stones must be classified. The stones marked by triangles must be classified alive. The stones marked by squares must be classified dead. The unmarked stones are irrelevant for scoring because they are not at the border of their area and their capturability does not affect the score. For example, the two black stones in the top-left corner kill the white block and are in Black's area. However, they can always be captured by White, so forcing them to be classified as alive or dead is misleading and even unnecessary. (The stones in the bottom left corner are alive in seki because neither side can capture. The two white stones in the upper right corner are adjacent to two neutral points and therefore also at the border of White's region.)

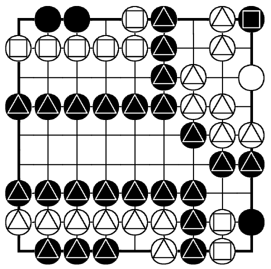
Fig. 1. Blocks to classify.

3.2 Recursion

Usually blocks of stones are not alive on their own. Instead they form chains or groups which are only alive in combination with other blocks. Their status also may depend on the status of neighbouring blocks of the opponent, i.e., blocks can live by capturing the opponent. (Although one might be tempted to conclude that life and death should be dealt with at the level of groups this does not really help because the human notion of a group is not well defined, difficult to program, and may even require an underlying notion of life and death.)

Because life and death of blocks is strongly related to the life and death of other blocks the status of other (usually nearby) blocks has to be taken into account. Partially this can be done by including features for nearby blocks in the representation. In addition, it seems natural to consider a recursive framework for classification which employs the predictions for other blocks to improve performance iteratively. In our implementation this is done by training a cascade of classifiers which use previous predictions for other blocks as additional input features.

4 Representation

In this section we will present the representation of blocks for classification. Several representations are possible and used in the field. The most primitive representations typically employ the raw board directly. A straightforward implementation is to concatenate three bitboards into a feature vector, for which the first bitboard contains the block to be classified, the second bitboard contains other friendly blocks, and the third bitboard contains the enemy blocks. Although this representation is complete, in the sense that all relevant information is preserved it is unlikely to be efficient because of the high dimensionality and lack of topological structure.

4.1 Features for Block Classification

A more efficient representation employs a set of features based on simple measurable geometric properties, some elementary Go knowledge and some hand-crafted specialised features. Several of these features are typically used in Go programs to evaluate positions [17,18]. The features are calculated for: (1) single friendly blocks, (2) single opponent blocks, (3) multiple blocks in chains, and (4) colour-enclosed regions (CERs).

For each block our representation consists of the following features (all features are single scalar values unless stated otherwise):

– *Size* measured in occupied points.
– *Perimeter* measured in number of adjacent points.
– *Opponents* are the occupied adjacent points.
– *(First-order) liberties* are the free (empty) adjacent points.
– *Protected liberties* are the liberties which normally should not be played by the opponent, because of suicide or being directly capturable (i.e., if the opponent plays there he has at most one liberty).
– *Auto-atari liberties* are liberties which by playing them reduce the liberties of the block from 2 to 1, which means that the block would become directly capturable (such liberties are protected for an adjacent opponent block).
– *Second-order liberties* are the empty points adjacent to but not part of the liberties.
– *Third-order liberties* are the empty points adjacent to but not part of the first-order and second-order liberties.
– *Number of adjacent opponent blocks*
– *Local majority* is the number of friendly stones minus the number of opponent stones within a Manhattan distance of 2 from the block.
– *Centre of mass* represented by the average distance of stones in the block to the closest and second-closest edge (using floating point scalars).
– *Bounding box size* is the number of points in the smallest rectangular box that can contain the block.

Adjacent to each block are colour-enclosed regions. CERs consist of connected empty and occupied points, surrounded by stones of one colour. (Notice that regions along the edge, such as an eye in the corner, are also enclosed). It is important to know whether an adjacent CER is fully accessible, because a fully accessible CER surrounded by safe blocks provides at least one sure liberty (the surrounding blocks are safe when they all have at least two sure liberties). To detect fully accessible regions we use so-called miai strategies as applied by Müller [14]. In addition to Müller's original implementation we (1) add miai-accessible interior empty points to the set of accessible liberties, and (2) use protected liberties for the chaining. An example of a fully accessible CER is shown in Figure 2. Here the idea is that if White plays on a marked empty point, Black replies on the other empty point marked by the same letter. By following this miai strategy Black is guaranteed to be able to occupy or become adjacent to all points in the region, i.e., all empty points in Figure 2 that are not directly adjacent to black stones are miai-accessible interior empty points; the points on the edge marked 'b' and 'e' were not used in Müller's original implementation [19]. Often it is not possible to find a miai strategy for the full region, in which case we call the CER partially accessible. In Figure 3 an example of a partially accessible CER is shown. In this case the 3 points marked 'x' form the inaccessible interior for the given miai strategy.

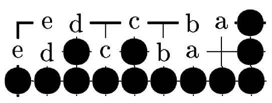

Fig. 2. Fully accessible CER.

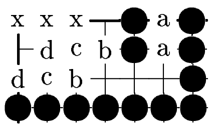

Fig. 3. Partially accessible CER.

Analysis of the CERs can provide us with several interesting features. However, the number of regions is not fixed, and our representation requires a fixed number of features. Therefore we decided to sum the features over all regions. For fully accessible CERs we include:

- *Number of regions*
- *Size* [3]
- *Perimeter*
- *Number of split points* in the CER. Split points are crucial points for preserving connectedness in the local 3×3 window around the point. (The region could still be connected by a big loop outside the local 3×3 window.) Examples are shown in Figure 4.

For partially accessible CERs we include:

- *Number of partially accessible regions*

[3] Since the regions may contain stones we deal with them as blocks of connected intersections regardless of the colour. Calculations of the various features, such as size, perimeter, and split points, are performed analogously to the calculations for normal blocks of stones of only one colour.

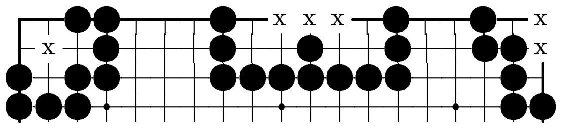

Fig. 4. Split points marked with x.

- *Accessible size*
- *Accessible perimeter*
- *Size* of the inaccessible interior.
- *Perimeter* of the inaccessible interior.
- *Split points* of the inaccessible interior.

Another way to analyse CERs is to look for possible eyespace. Points forming the eyespace should be empty or contain capturable opponent stones. Empty points directly adjacent to opponent stones are not part of the eyespace. Points on the edge with one or more diagonally adjacent alive opponent stones and points with two or more diagonally adjacent alive opponent stones are false eyes. False eyes are not part of the eyespace (we ignore the unlikely case where a big loop upgrades false eyes to true eyes). For example, in Figure 5 the points marked ‘e’ belong to Black’s eyespace and the points marked ‘f’ are false eyes for White. Initially we assume all diagonally adjacent opponent stones to be alive. However, in the recursive framework (see below) the eyespace is updated based on the status of the diagonally adjacent opponent stones after each iteration.

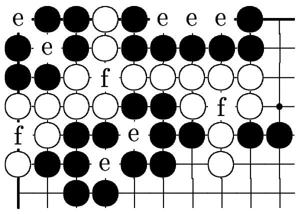

Fig. 5. True and false eyespace.

For directly adjacent eyespace of the block we include:

- *Size*
- *Perimeter*

Since we are dealing with final positions it is often possible to use the optimistic assumption that all blocks with shared liberties can form a chain (during the game this assumption is dangerous because the chain may be split). Examples of a black and a white optimistic chain are shown in Figure 6. For this, so-called, optimistic chain we include:

- *Number of blocks*
- *Size*
- *Perimeter*
- *Split points*

– *Number of adjacent CERs*
– *Number of adjacent CERs with eyespace*
– *Number of adjacent CERs, fully accessible* from at least one block.
– *Size of adjacent eyespace*
– *Perimeter of adjacent eyespace*. (Again, in the case of multiple connected regions for the eyespace, size and perimeter are summed over all regions.)
– *External opponent liberties* are liberties of adjacent opponent blocks which are not accessible from the optimistic chain.

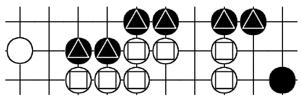

Fig. 6. Marked optimistic chains.

Adjacent to the block in question there may be opponent blocks. For the weakest (measured by the number of liberties) directly adjacent opponent block we include:

– *Perimeter*
– *Liberties*
– *Shared liberties*
– *Split points*
– *Perimeter of adjacent eyespace*

The same features are also included for the second-weakest directly adjacent opponent block and the weakest opponent block directly adjacent to or sharing liberties with the optimistic chain of the block in question (so the weakest directly adjacent opponent block may be included twice).

By comparing a flood fill starting from Black with a flood fill starting from White we find unsettled empty regions which are disputed territory (assuming all blocks are alive). If the block is adjacent to disputed territory we include:

– *Direct liberties* in disputed territory.
– *Liberties of all friendly blocks* in disputed territory.
– *Liberties of all enemy blocks* in disputed territory.

4.2 Additional Features for Recursive Classification

For the recursive classification we use the predicted values of previous classifications, which are floating point scalars in the range between 0 (dead) and 1 (alive), to construct the following six additional features:

– *Predicted value* of the strongest friendly block with a shared liberty.
– *Predicted value* of the weakest adjacent opponent block.
– *Predicted value* of the second-weakest adjacent opponent block.

– *Average predicted value* of the weakest opponent block's optimistic chain.
– *Adjacent eyespace size* of the weakest opponent block's optimistic chain.
– *Adjacent eyespace perimeter* of the weakest opponent block's optimistic chain.

Next to these additional features the predictions are also used to update the eyespace, i.e., dead blocks can become eyespace for the side that captures, alive blocks cannot provide eyespace, and diagonally adjacent dead opponent stones are not counted for detecting false eyes.

5 The Data Set

In the experiments we used game records obtained from the NNGS archive [13]. All games were played on the 9×9 board between 1995 and 2002. We only considered games which are played to the end and scored, thus ignoring unfinished or resigned games. Since the game records only contain a single numeric value for the score, we had to find a way to label all blocks.

5.1 Scoring the Data Set

For scoring the dataset we initially used a combination of GNUGO (version 3.2) [20] and manual labelling. Although GNUGO has the option to finish games and label blocks the program could not be used without human supervision. The reasons for this are threefold: (1) bugs, (2) the inherent complexity of the task, and (3) the mistakes made by weak human players who ended the game in positions that were not final, or scored them incorrectly. Fortunately, nearly all mistakes were easily detected by comparing GNUGO's scores and labelled boards with the numeric scores stored in the game records. As an extra check all boards containing open regions with unsettled interior points (where flood filling does not give the same result as distance-based scoring) were also inspected manually.

Since the scores did not match in many positions the labelling proved to be very time consuming. We therefore only used GNUGO to label the games played in 2002 and 1995. With the 2002 games a classifier was trained. When we tested the performance on the 1995 games it outperformed GNUGO's labelling. Therefore our classifier replaced GNUGO for labelling the other games (1996-2001), retraining it each time a new year was labelled. Although this sped up the process it still required a fair amount of human intervention mainly because of games that contained incorrect scores in their game record. A few hundred games had to be thrown out completely because they were not finished, contained illegal moves, contained no moves at all (for at least one side), or both sides were played by the same player. In a small number of cases, where the last moves would have been trivial but not actually played, we made the last few moves manually.

Eventually we ended up with a dataset containing 18,222 final positions. Around 10% of these games were scored incorrectly (by the players) and were inspected manually. (Actually the number of games we inspected is significantly higher because of the games that were thrown out and because both our initial classifiers and GnuGo made mistakes.) On average the final positions contained 5.8 alive blocks, 1.9 dead blocks, and 2.7 irrelevant blocks. (In the case that one player gets the full board we count all blocks of this player as irrelevant, because there is no border. Of course, in practice at least one block should be classified as alive, which appears to be learned automatically without any special attention.)

Since the Go scores on the 9×9 board range from −81 to +81 the chances of an incorrect labelling leading to a correct score are low, nevertheless it could not be ruled out completely. On inspecting an additional 1% of the positions randomly we found none that were labelled incorrectly. Finally, when all games were labelled, we re-inspected all positions for which our best classifier seemed to predict an incorrect score. This final pass detected 42 positions (0.2%) which were labelled incorrectly, mostly because our initial classifiers had made the same mistakes as the players who scored the games.

5.2 Statistics

Since many game records contained incorrect scores we looked for reasons and gathered statistics. The first thing that came to mind is that weak players might not know how to score. Therefore in Figure 7 the percentage of incorrectly scored games related to the strength of the players is shown. (Although in each game only one side may have been responsible for the incorrect score, we always assigned blame to both sides.) The two marker types distinguish between rated and unrated players. Although unrated players have a value for their rating, it is an indication given by the player and not by the server. Only after playing sufficiently many games the server assigns players a rating.

Although a significant number of games are scored incorrectly this is usually not considered a problem when the winner is correct. (Players typically forget to remove some stones when they are far ahead.) Figure 8 shows how often incorrect scoring by rated players converts a loss into a win (cheater) or a win into a loss (victim).

It should be noted that the percentages in Figures 7 and 8 were weighted over all games, regardless of who was the player. Therefore, they do not necessarily reflect the probabilities for individual players, i.e., the statistics can be dominated by a small group of players that played many games. This group at least contains some computer players which have a tendency to get robbed of their points in the scoring phase. Hence, we calculated some statistics that were normalised over individual players, e.g., statistics of players who played hundreds of games were

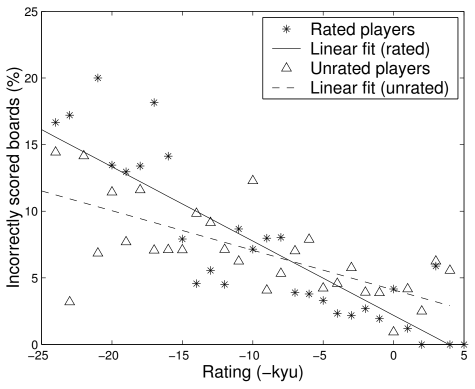

Fig. 7. Incorrect scores.

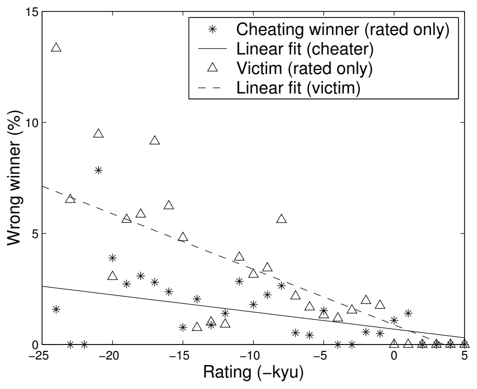

Fig. 8. Incorrect winners.

weighted equal to the statistics of players who played only a small number of games. Thereupon we found that for rated players the average probability of scoring a game incorrectly is 4.2%, the probability of cheating (the incorrect score converts a loss into a win) is 0.66%, and the probability of getting cheated is 0.55%. For unrated players the average probability of scoring a game incorrectly is 11.2%, the probability of cheating is 2.1%, and the probability of getting cheated is 1.1%. The fact that, when we normalise over players, the probability of getting cheated is lower than the probability of cheating is the result of a small group of players (several of them are computer programs) who systematically lose points in the scoring phase, and a larger group of players who take advantage of that fact.

6 Experiments

In this section experimental results are presented for: (1) selecting a classifier, (2) performance of the representation, (3) recursive performance, (4) full-board performance, and (5) performance on the 19×19 board. Unless stated otherwise the various training and validation sets, used in the experiments, were extracted from games played between 1996 and 2002. The test set was always the same, containing 7149 labelled blocks extracted from 919 games played in 1995.

6.1 Selecting a Classifier

An important choice is selecting a good classifier. In pattern recognition there is a wide range of classifiers to choose from [21]. We tested a number of well-known classifiers for their performance (without recursion) on datasets of 100, 1000, and 10,000 examples. The classifiers are: Nearest Mean Classifier (NMC), Linear Discriminant Classifier (LDC), Logistic Linear Classifier (LOGLC), Quadratic Discriminant Classifier (QDC), Nearest Neighbour Classifier (NNC), K-Nearest Neighbours Classifier (KNNC), BackPropagation Neural net Classifier with momentum and adaptive

Table 1
Performance of classifiers without recursion.

Classifier	Training size	Training error	Test error	Training time	Classi. speed
		(%)	(%)	(s)	(s^{-1})
NMC	100	2.8	3.9	0.0	4.9×10^4
	1000	4.0	3.8	0.1	5.2×10^4
	10,000	3.8	3.6	0.5	5.3×10^4
LDC	100	0.7	3.0	0.0	5.1×10^4
	1000	2.1	2.0	0.1	5.2×10^4
	10,000	2.2	1.9	0.9	5.3×10^4
LOGLC	100	0.0	9.3	0.2	5.2×10^4
	1000	0.0	2.6	1.1	5.2×10^4
	10,000	1.0	1.2	5.6	5.1×10^4
QDC	100	0.0	13.7	0.1	3.1×10^4
	1000	1.0	2.1	0.1	3.2×10^4
	10,000	1.9	2.1	1.1	3.2×10^4
NNC	100	0.0	18.8	0.0	4.7×10^3
	1000	0.0	13.5	4.1	2.4×10^2
	10,000	0.0	10.2	4.1×10^3	2.4×10^0
KNNC	100	7.2	13.1	0.0	4.8×10^3
	1000	4.2	4.4	1.0×10^1	2.4×10^2
	10,000	2.8	2.8	9.4×10^3	2.6×10^0
BPNC	100	0.5	3.6	2.9	1.8×10^4
	1000	0.2	1.5	1.9×10^1	1.8×10^4
	10,000	0.5	1.0	1.9×10^2	1.9×10^4
LMNC	100	2.2	7.6	2.6×10^1	1.8×10^4
	1000	0.7	2.8	3.2×10^2	1.8×10^4
	10,000	0.5	1.2	2.4×10^3	1.9×10^4
RPNC	100	1.5	4.1	1.4	1.8×10^4
	1000	0.2	1.7	7.1	1.8×10^4
	10,000	0.4	1.1	7.1×10^1	1.9×10^4

learning (BPNC), Levenberg-Marquardt Neural net Classifier (LMNC), and RProp Neural net Classifier (RPNC). Some preliminary experiments with a Support Vector Classifier, Decision Tree Classifiers, a Parzen classifier, and a Radial Basis Neural net Classifier were not pursued further because of excessive training times and/or poor

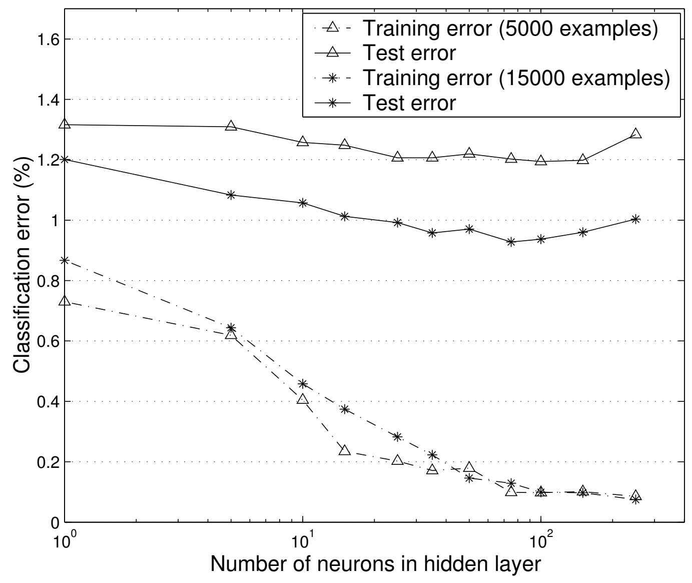

Fig. 9. Sizing the neural network for the RPNC.

performance. All classifiers except the neural net classifiers, for which we directly used the standard Matlab toolbox, were used as implemented in PRTools3 [22].

The results, shown in Table 1, indicate that performance first of all depends on the size of the training set. The linear classifiers perform better than the quadratic classifier and nearest neighbour classifiers. For large datasets training KNNC is very slow because it takes a long time to find an optimal value of the parameter k. The number of classifications per second of (K)NNC is also low because of the large number of distances that must be computed (all training examples are stored). Although the performance of the nearest neighbour classifiers might be improved by editing and condensing the dataset, we did not investigate them further.

The best classifiers are the neural network classifiers. It should however be noted that their performance may be slightly overestimated with respect to the size of the training set, because we used an additional validation set to stop training (this was not possible for the other classifiers because they are not trained incrementally). The Logistic Linear Classifier performs nearly as well as the neural network classifiers, which is quite an achievement considering that it is just a linear classifier.

The results of Table 1 were obtained with networks that employed one hidden layer containing 15 neurons with hyperbolic tangent sigmoid transfer functions. Since our choice for 15 neurons was quite arbitrary a second experiment was performed in which we varied the number of neurons in the hidden layer. In Figure 9 results are shown for the RPNC. The classification errors marked with triangles represent results for training on 5000 examples, the stars indicate results for training on 15,000 examples. The solid lines are measured on the independent test set, whereas the dash-dotted lines are obtained on the training set. The results show that even moderately sized networks easily overfit the data. Although the performance initially improves with the size of the network, it seems to level off for networks with over 50 hidden neurons (the standard deviation is around 0.1%). Again, the key factor in improving performance clearly is in increasing the size of the training set.

6.2 Performance of the Representation

In section 4 we claimed that a raw board representation is inefficient for predicting life and death. To validate this claim we measured the performance of such a representation and compared it to our specialised representation.

The raw representation consists of three concatenated bitboards, for which the first bitboard contains the block to be classified, the second bitboard contains other friendly blocks, and the third bitboard contains the enemy blocks. To remove symmetry the bitboards are rotated such that the centre of mass of the block to be classified is always in a single canonical region.

Since high-dimensional feature spaces tend to raise several problems which are not directly caused by the quality of the individual features we also tested two compressed representations. These compressed representations were generated by performing Principal Component Analysis (PCA) on the raw representation. For the first PCA mapping the number of features was chosen identical to our specialised representation. For the second PCA mapping the number of features was set to preserve 90% of the total variance.

The results, shown in Table 2, are obtained for the RPNC with 15, 35, and 75 neurons

Table 2
Performance of the raw representation.

Training Size	Extractor	Test error 15 neurons (%)	Test error 35 neurons (%)	Test error 75 neurons (%)
100	-	29.1	26.0	27.3
100	pca1	22.9	22.9	22.3
100	pca2	23.3	24.3	21.9
1000	-	13.7	13.5	13.4
1000	pca1	16.7	16.2	15.6
1000	pca2	14.2	14.5	14.4
10,000	-	7.5	6.8	6.5
10,000	pca1	9.9	9.3	9.1
10,000	pca2	8.9	8.2	7.7

in the hidden layer, for training sets with 100, 1000, and 10,000 examples. All values are averages over 11 runs with different training sets, validation sets (same size as the training set), and random initialisations. The errors, measured on the test set, indicate that a raw representation alone requires too many training examples to be useful in practice. Even with 10,000 training examples the raw representation performs much more weakly than our specialised representation with only 100 training examples. Simple feature-extraction methods such as Principal Component Analysis do not seem to improve performance, indicating that preserved variance of the raw representation is relatively insignificant for determining life and death.

6.3 Recursive Performance

Our recursive framework for classification is implemented as a cascade of classifiers which use extra features, based on previous predictions as discussed in subsection 4.2, as additional input. The performance measured on an independent test set for the first 4 steps is shown for various sizes of the training set in Table 3. The results are averages of 5 runs with randomly initialised networks containing 50 neurons in the hidden layer (the standard deviation is around 0.1%).

Table 3
Recursive performance.

Training Size	Direct error (%)	2-step error (%)	3-step error (%)	4-step error (%)
1000	1.93	1.60	1.52	1.48
10,000	1.09	0.76	0.74	0.72
100,000	0.68	0.43	0.38	0.37

The results show that recursive predictions improve the performance. However, the only significant improvement comes from the first iteration. The improvements are far from significance for the average 3- and 4-step errors. The reason for this is that sometimes the performance got stuck or even worsened after the first iteration. Preliminary experiments suggest that large networks were more likely to get stuck after the first iteration than small networks, which might indicate some kind of overfitting. A possible solution to overcome this problem is to retrain the networks a number of times, and pick the best based on the performance on the validation set. If we do this our best networks, trained on 100,000 training examples, achieve a 4-step error of 0.25%. We refer to the combination of the four cascaded classifier networks and the marking of empty intersections based on the distance to the nearest living block (which may be verified by comparing to flood filling, see section 2) by CSA* (Cascaded Scoring Architecture).

In Figure 10 we show twelve examples of mistakes that are made by direct classification without recursion, which can be corrected by using the 4-step recursion of

CSA*. All marked blocks were initially classified incorrectly. Initially, the blocks marked with squares were classified as alive, and the blocks marked with triangles were classified as dead. After recursion this was corrected so that the blocks marked with squares are classified as dead, and the blocks marked with triangles are classified as alive.

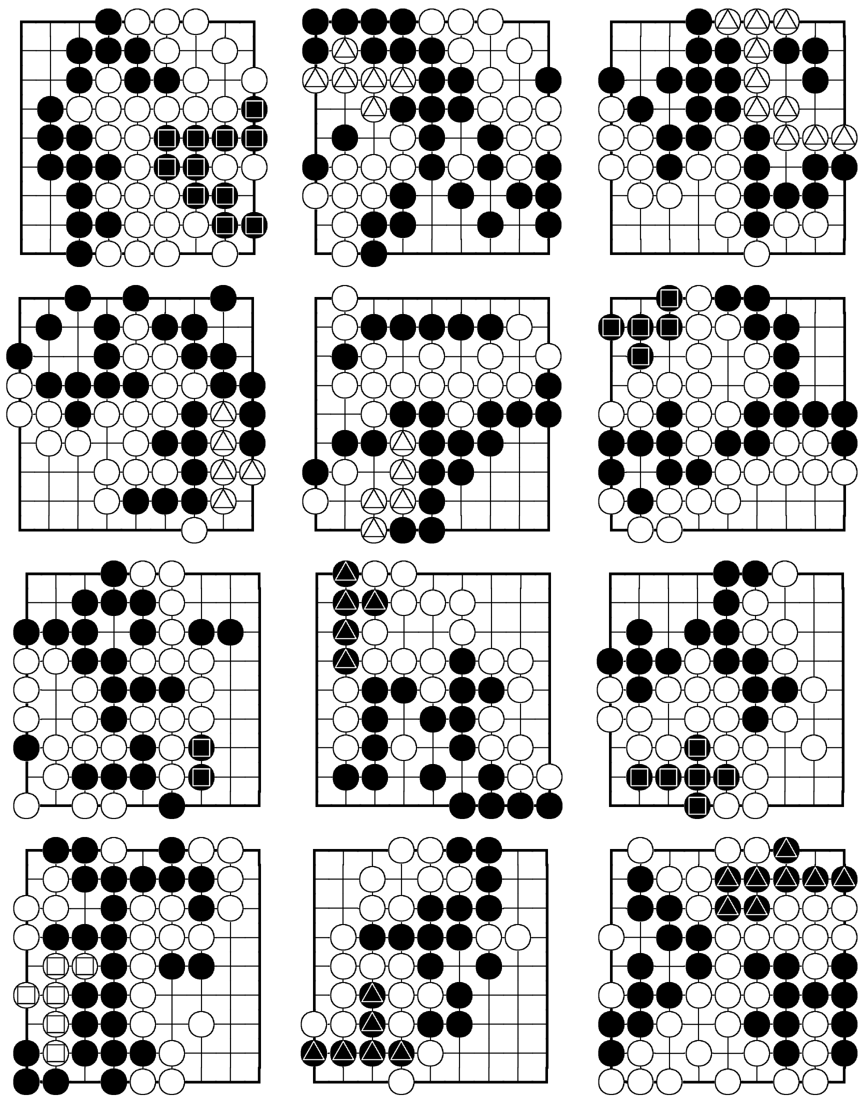

Fig. 10. Examples of mistakes that are corrected by recursion.

6.4 Full-Board Performance

So far we have concentrated on the percentage of blocks that are classified correctly. Although this is an important measure it does not directly indicate how often boards will be scored correctly (a board may contain multiple incorrectly classified blocks). Further, we do not yet know what the effect is on the score in number of board points. Therefore we tested our classifiers on the full-board test positions, which were not

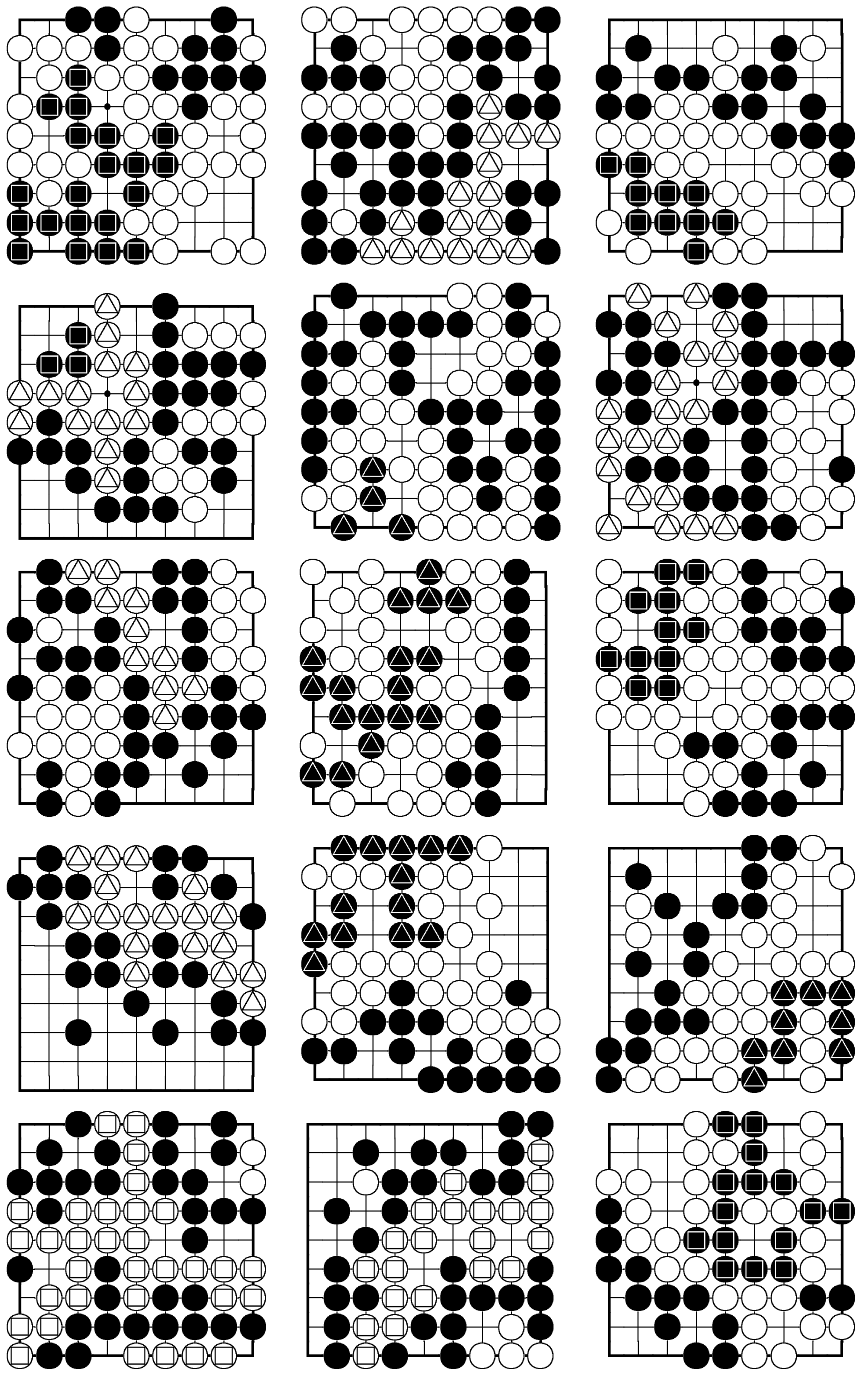

Fig. 11. Examples of incorrectly scored positions.

used for training or validation.

For CSA* we found that 1.1% of the boards were scored incorrectly. For 0.5% of the boards the winner was not identified correctly. The average number of incorrectly scored board points (using distance-based scoring) was 0.15. However, in case a board is scored incorrectly it usually affects around 14 board points (which counts double in the numeric score).

In Figure 11 we show examples of the (rare) mistakes that are still made by the 4-step classification of CSA*. All marked blocks were classified incorrectly. The blocks marked with squares were incorrectly classified as alive. The blocks marked with triangles were incorrectly classified as dead. The difficult positions typically include seki, long chains connected by false eyes, bent four and similar looking shapes, and rare shapes such as ten-thousand year ko. In general we believe that many of these mistakes can be corrected by adding more training examples. However, for some positions it might be best to add new features or use a local search.

6.5 Performance on the 19×19 Board

The experiments presented above were all performed on the 9×9 board which, as was pointed out before, is a most challenging environment. Nevertheless, it is interesting to test whether (and if so to what extent) the techniques scale up to the 19×19 board. So far we did not focus on labelling large quantities of 19×19 games. Therefore, training directly on the 19×19 board was not an option. Despite of this we tested CSA*, which was trained using blocks observed on the 9×9 board, on the problem set *IGS_31_counted* from the Computer Go Test Collection. This set contains 31 labelled 19×19 games played by amateur dan players, and was used by Müller [14]. On the 31 final positions our 4-step classifier classified 5 blocks incorrectly (0.5% of all relevant blocks), and as a consequence 2 final positions were scored incorrectly. The average number of incorrectly scored board points was 2.1 (0.6%).

In his article Müller [14] stated that the heuristic classification by his program EXPLORER classified most blocks correctly. Although we do not know the exact performance of EXPLORER we believe it is safe to say that CSA*, which classified 99.5% of all blocks correctly, is performing at least at a comparable level. Furthermore, since our system was not trained explicitly for 19×19 games there may still be significant room for improvement.

7 Conclusion

We have developed a Cascaded Scoring Architecture (CSA*) that learns to score final positions from labelled examples. On unseen game records our system scored around 98.9% of the positions correctly without any human intervention. Compared to the average rated player on NNGS, who has a rating of 7 kyu for scored 9×9 games, we may conclude that CSA* is more accurate at removing all dead blocks, and performs comparably on determining the correct winner.

By comparing numeric scores and counting unsettled interior points nearly all incorrectly scored final positions can be detected (for verification by a human operator). Although some final positions are assessed incorrectly by our classifier, most are in fact scored incorrectly by the players. Detecting games that were incorrectly scored by the players is important because most machine-learning methods require reliable training data for a good performance.

7.1 On-going Work

By providing reliable score information CSA* opens the large source of Go knowledge which is implicitly available in human game records. The next step is to apply machine learning in non-final positions. The representation, techniques, and the data set presented in this article provide a solid basis for static predictions in non-final positions. Recent work shows promising results for predicting life and death [23] as well as predicting territory [24] during the game.

So far, the good performance of CSA* was obtained without any search, indicating that static evaluation is sufficient for most human final positions. Nevertheless, we believe that some (selective) search can still improve the performance. Adding selective features that involve search and integrating our system into MAGOG, our 9×9 Go program, will be an important next step.

Although the performance of CSA* is already quite good for labelling game records, there are, at least in theory, still positions which may be scored incorrectly when the classifiers make the same mistakes as the human players. Future work should determine how often this happens in practice.

We are considering to use our learning framework to extract human readable heuristic rules about life and death. By formulating the heuristics as lemmas and proving them by using domain specific knowledge, we may be able to increase the scope of the provably correct scoring method for solving Go positions presented in [25].

Acknowledgements

The work is funded by the Netherlands Organisation for Scientific Research (NWO), dossier number 612.052.003.

References

[1] J. van der Steen, Gobase.org - Go games, Go information and Go study tools (2003).
URL `http://gobase.org/`

[2] D. Lichtenstein, M. Sipser, Go is polynomial-space hard, J. ACM 27 (2) (1980) 393–401.

[3] J. M. Robson, The complexity of Go, in: IFIP World Computer Congress 1983, 1983, pp. 413–417.

[4] B. Bouzy, T. Cazenave, Computer Go: An AI oriented survey, Artificial Intelligence 132 (1) (2001) 39–102.

[5] M. Müller, Computer Go, Artificial Intelligence 134 (1-2) (2002) 145–179.

[6] G. Tesauro, Temporal difference learning and TD-Gammon, Communications of the ACM 38 (3) (1995) 58–68.
URL `http://www.research.ibm.com/massdist/tdl.html`

[7] F. Dahl, Honte, a Go-playing program using neural nets, in: J. Fürnkranz, M. Kubat (Eds.), Machines that Learn to Play Games, Nova Science Publishers, Huntington, NY, 2001, Ch. 10, pp. 205–223.

[8] M. Enzenberger, The integration of a priori knowledge into a Go playing neural network, unpublished manuscript (September 1996).
URL `http://home.t-online.de/home/markus.enzenberger/neurogo.ps.gz`

[9] N. Schraudolph, P. Dayan, T. Sejnowski, Temporal difference learning of position evaluation in the game of Go, in: J. D. Cowan, G. Tesauro, J. Alspector (Eds.), Advances in Neural Information Processing 6, Morgan Kaufmann, San Francisco, CA, 1994, pp. 817–824.
URL `ftp://ftp.idsia.ch/pub/nic/nips93.ps.gz`

[10] E. C. D. van der Werf, Aya wins 9×9 Go tournament, ICCA Journal 26 (4) (2003) 263.

[11] H. Enderton, The Golem Go program, Tech. Rep. CMU-CS-92-101, School of Computer Science, Carnegie-Mellon University (December 1991).
URL `ftp://reports.adm.cs.cmu.edu/usr/anon/1992/CMU-CS-92-101.ps`

[12] E. C. D. van der Werf, J. W. H. M. Uiterwijk, E. O. Postma, H. J. van den Herik, Local move prediction in Go, in: J. Schaeffer, M. Müller, Y. Björnsson (Eds.), Computers and Games: Third International Conference, CG 2002, Edmonton, Canada, July 2002: revised papers, Vol. 2883 of LNCS, Springer-Verlag, 2003, pp. 393–412.

[13] NNGS, The no name Go server game archive (2002).
URL `http://nngs.cosmic.org/gamesearch.html`

[14] M. Müller, Playing it safe: Recognizing secure territories in computer Go by using static rules and search, in: H. Matsubara (Ed.), Proceedings of the Game Programming Workshop in Japan ’97, Computer Shogi Association, Tokyo, Japan, 1997, pp. 80–86.

[15] D. B. Benson, Life in the game of Go, Information Sciences 10 (1976) 17–29, reprinted in Computer Games, Levy, D.N.L (editor), Vol. II, pp. 203-213, Springer-Verlag, New York, 1988. ISBN 0-387-96609-9.

[16] S. T. Dekker, H. J. van den Herik, I. S. Herschberg, Complexity starts at five, ICCA Journal 10 (3) (1987) 125–138.

[17] K. Chen, Z. Chen, Static analysis of life and death in the game of Go, Information Sciences 121 (1999) 113–134.

[18] D. Fotland, Static eye analysis in 'THE MANY FACES OF GO', ICGA Journal 25 (4) (2002) 201–210.

[19] M. Müller, personal communication (2003).

[20] Free Software Foundation, Gnu Go (2003).
URL `http://www.gnu.org/software/gnugo/gnugo.html`

[21] A. K. Jain, R. P. W. Duin, J. Mao, Statistical pattern recognition: A review, IEEE Transactions on Pattern Analysis and Machine Intelligence 22 (1) (2000) 4–37.

[22] R. Duin, PRTools, a Matlab Toolbox for Pattern Recognition (2000).
URL `http://www.ph.tn.tudelft.nl/prtools/`

[23] E. C. D. van der Werf, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, Learning to predict life and death from Go game records, Information Sciences, accepted for publication (2004).

[24] E. C. D. van der Werf, H. J. van den Herik, J. W. H. M. Uiterwijk, Learning to estimate potential territory in the game of Go, in: Proceedings of the 4th International Conference on Computers and Games (CG'04) (Ramat-Gan, Israel, July 5-7), 2004, to appear in LNCS, Springer-Verlag, Berlin.

[25] E. C. D. van der Werf, H. J. van den Herik, J. W. H. M. Uiterwijk, Solving Go on small boards, ICGA Journal 26 (2) (2003) 92–107.